Are Your Cookies Telling Your Fortune?

An analysis of weak cookie secrets and OSINT

May 2018
Table of Contents

1. Overview .................................................................................................................................................. 3
2. What Is a “Cookie Secret”? ...................................................................................................................... 4
3. Gathering Potential Targets and Secrets .................................................................................................. 6
4. Cracking The Secrets ............................................................................................................................... 8
5. An Example of an Attack on Passport.js Using OpenID ......................................................................... 12
6. An Example of an Attack on Passport.js Using Local Authentication ..................................................... 18
7. Summary & Recommendations ................................................................................................................ 22
8. References ................................................................................................................................................ 23
9. Appendix .................................................................................................................................................. 24
   9.1. Shodan CSV to JSON Script ................................................................................................................ 24
   9.2. Modifications to app.js ...................................................................................................................... 25
   9.3. Contents of new_cookie.json ........................................................................................................... 26
1. Overview

With the exponential growth the web has witnessed in the last decade, both in terms of services and technical sophistication, many new technologies have come to the aid of developers in order to develop better products, more efficiently.

As new libraries and frameworks begin to gain momentum and establish themselves as the new de-facto standards, experienced and new developers alike, will frequently need to revisit “the basics”.

One such example, and the focus of this paper, is the use of sessions to introduce a persistent state over HTTP. Although securing a session can be done with ease, it is not uncommon to find educational material or public forums which fail to explain the importance of using unique and strong session secrets.

This paper aims to provide an analysis of Node.js applications, using information gathered solely through open-source intelligence, as to whether developers are following the best practises, as well as the trends in those that have not.

In particular, the target of the analysis will be Node.js applications that have been built using the cookie-session middleware. As of May, 2018, the cookie-session package had been downloaded an average of 205,000 times per month [1]; indicating its usage is quite wide spread.
2. What Is a "Cookie Secret"?

First and foremost, it's important to understand the problem that signed cookies attempt to (and for the most part) solve.

Cookies have traditionally been something developers have avoided, if the integrity of the data to be stored must be maintained. The reason for this, is that cookies are sent to the server as plain-text HTTP headers; meaning a malicious actor can edit them with ease.

In the code seen in illustration 1, the server will check if the isAdmin cookie equals true, and if so, will assume the user is indeed an admin. Bypassing this check would be as trivial as an attacker including 'Cookie: isAdmin=true' in the HTTP request.

```javascript
1 const express = require('express')
2 const cookieParser = require('cookie-parser')
3
4 let app = express()
5 app.use(cookieParser())
6
7 app.get('/', function (req, res, next) {
8     if (req.cookies['isAdmin'] === 'true') {
9         res.send('Mr. Anderson, welcome back.')
10     } else {
11         res.status(401).send('Nope.')
12     }
13 })
14
15 app.listen(3000)
```

Illustration 1: An example of an insecure use of cookies

The solution to this problem, is to "sign" the cookies when sending them to the client and verify the signature when they are sent back from the client in subsequent requests.
The signing process consists of taking the data being sent in the cookie and then using a hashing algorithm to hash a combination of the data and the secret. If the hash generated by the server matches the one sent by the client, then the server can be [relatively] confident that the cookie has not been tampered with.

Should an attacker be able to guess the secret, or acquire it, the integrity of the entire process is voided, as they would be capable of generating valid signatures by following the same signature signing process as the server they are attacking.
3. Gathering Potential Targets and Secrets

Web applications that make use of the cookie-session middleware use a consistent naming convention for the session cookie's signature. When initialising the middleware, one of the options is an optional cookie name, which defaults to "session".

Once a session is initialised and the cookie is sent to the user, an additional cookie containing the signature will be sent. The signature cookie will be named "{name}.sig", where {name} is the name specified during initialisation.

With this naming convention in mind, searching Shodan for "session.sig" returned 8,190 hosts that were, seemingly, sending signature cookies generated by the cookie-session middleware. The results of this search were subsequently exported to a JSON file for a more in depth analysis later.

The next set of data that needed to be collected, in order to perform the analysis, was a list of potential secrets. Using open-source intelligence, a list was compiled, containing 289 unique secrets. One of the sources used, which yielded a large amount of the secrets, was GitHub.

Searching GitHub for the term "secret:" to find code that assigns a value to a property named "secret", yielded 1,979,689 code results when filtered to JavaScript code. Across all languages, there were over 20,000,000 results, but that number is to be taken lightly, as the syntax searched
for would be invalid in various other languages. For example, there were 1,644,563 results in HTML code - but without some very unique design decisions, there would not be any secret definitions within HTML files.

In addition to GitHub, a variety of online tutorials and project documentation, such as Express’ best practices guide [2], were used to gather more potential common secrets, which may have been copied and pasted into production code, due to no emphasis being placed on the importance of the secret’s entropy.
4. Cracking The Secrets

To aid in the testing of the hosts harvested from Shodan, a small utility was developed - Cookie Monster [3]. Cookie Monster will take a JSON file containing an array of objects that contain the following pieces of data for a host:

- IP address
- Port number
- Session cookie data
- Session cookie signature
- The name of the session cookie

As the extract downloaded from Shodan was excessively verbose, and did not follow this format, some preparation was required. The first step in preparing the data was to use the Shodan command-line interface [4] to parse the extract into a CSV file, by running:

```sh
shodan parse --fields ip_str,port,data --separator '||' session.sig.json.gz > servers.csv
```

Once in CSV format, the file was processed into JSON, using the script found in section 9.1.

Illustration 4: The normalised Shodan data ready for use with Cookie Monster
After launching Cookie Monster with the newly created file in batch mode, a total of 62 cookies were successfully found from a data set consisting of 8,186 samples.

Illustration 5: Cookie Monster successfully finding various cookie secrets

Illustration 6: A sample of the results output by Cookie Monster
The most common secret within the results was "secret", with a total of 45 instances, followed by "keyboard cat" with 11 instances - both of which, are secrets that were found in a number of online tutorials.

![Illustration 7: The secrets identified by Cookie Monster](image)

Of the 62 servers found to be vulnerable to cookie manipulation, 16 of those were utilising Passport.js for authentication; identified by the presence of the "passport" object in the cookie.

![Illustration 8: A sample of a host utilising Passport.js](image)

```json
{
    "name": "session",
    "data": "eyJwYXNzcG9ydCI6IjEzNiIsInN5cG9zcyI6ImNvbSIsInN1YiI6IjIwMjM4NDAwMjIzNCIsInNjb3BlIjoiY29tcHJvY2F0aW9uIiwic2NhbGUiOltdLCJzdWIiOiIxNTM0Mjg1NDkyMzAiLCJpYXQiOiJrdWx0b2xpbmcifQ==",
    "sig": "eyJ0eXBlIjpbIHNsb3NlZCBXYXNzIiB7ImNyb3NzIjoiQnVpbmdzIiB7ImludG9za1wiOlt7InN0eWxlIjoiaHR0cHM6Ly90ZXN0LmRldmVsdGVyLmNuL2Jvc3QvdG8vd3Avc2FtcGFuZ3Vzcy90b2tlbi90d2lkdGgvd3d3LzIwMDAwMDMyMC0xMjI5LDAsMTgwMy0wMS0yMC0xMCJ9fX0=
    "ip": "192.168.1.1",
    "port": "443",
    "decodedData": "{\"passport\":{}}",
    "secret": "secret"
}
```
Passport.js is authentication middleware for Node.js, providing a range of authentication strategies, including Facebook, Twitter and Google authentication [5]. As these servers are using Passport.js in combination with the cookie-session middleware, there is a significantly high chance that they are vulnerable to an authentication bypass and potentially privilege escalation.

Due to the standardised schemas, if an attacker can be sure of which strategy is being used, they would be able to either forge an entirely new cookie and impersonate other users / bypass authentication, or alter an existing one.

In addition to Passport.js, other common objects found within the results were:

- CSRF tokens, believed to be generated by csurf [6]
- Flash messages, most likely generated by Express' Flash [7] middleware

Neither of these objects offer any value to a would-be attacker. The CSRF tokens are implemented as a way of preventing an attacker automating an action on a user’s behalf, meaning there is no valid attack vector. The flash messages are generated by the server in order to be passed to the client, not vice versa, meaning in most instances, any values within this cookie sent back to the server are likely to be ignored.
5. An Example of an Attack on Passport.js Using OpenID

To demonstrate how easily an attack of this nature can be carried out, an environment was setup which simulated that of the vulnerable hosts we had identified (i.e. using Passport.js as an authentication method).

The Passport strategy used for the test lab was the Passport-Steam strategy, which allows users registered with the Steam gaming platform to login using OpenID 2.0. The vendor of this strategy provides an example project [8] which was used as the basis of the test.

Some minor modifications were made to the `app.js` file to introduce the use of the cookie-session middleware. The specific changes made to the file can be found in the diff found in section 9.2.

Initially accessing the web application presents the user with two links - one to view their account information, and one to login with. Should the user attempt to access the account information without having a valid session, they will be redirected to the home page.

![Illustration 9: The home page of the passport-steam example](image-url)
Clicking the "Sign On" link begins the authentication process, and redirects the user to the Steam website, allowing them to login using their Steam credentials.

Illustration 10: The Steam login page

Should the credentials be entered correctly, the user will subsequently be redirected back to the sample application's home page; where it is now possible to view the account details.

Illustration 11: The details of the Steam account used for authentication

ID: 76561196833340125
Name: Rob @ Digital Interruption
Are Your Cookies Telling Your Fortune?

The ID number seen in illustration 11 is the **public** ID number of the Steam account that was used to login with, which can be seen in the URL of the account's profile page.

As the ID numbers can be harvested publicly, it'd be possible to authenticate as any Steam user without valid credentials, providing the cookie could be successfully modified; emphasising, again, the importance of strong secrets.

Examining the response of the request made after successful authentication will reveal that the server sends the base64 encoded session cookie and signature back to the client.
Decoding the value of **session** reveals all the data gathered from Steam that is used to identify the user.

The data within the session cookie consists of:

- **provider** - the name of the provider / strategy used to authenticate.
- **json** - a copy of the JSON data returned from Steam.
- **id** - the user’s public steam ID.
- **displayName** - the display name used on Steam.
- **photos** - an array of the user’s profile pictures / avatars.
- **identifier** - the open ID identifier URL for the user’s account.

The property that is of particular interest within this cookie, is the **id** property. Modifying this, will achieve the previously discussed goal of impersonating other Steam users, without the need for the account credentials.

For the purposes of the test, the modified contents of the decoded cookie were placed in a file, which can be found included in section 9.3; the modifications made are highlighted in red.

After re-encoding the file in the Decoder tab of Burp, the new base64 string was assigned as the value of the **session** cookie in the Repeater tab. Upon executing the request, however, the response code was 302, redirecting back to the home page; i.e. the behaviour previously witnessed if the user tries to view the account details page without a valid session.
Are Your Cookies Telling Your Fortune?

Although the secret of the sample application is already known, in this instance, for further proof of concept, the original session cookie and signature were processed with Cookie Monster to verify the secret is identified; confirming it to be "your secret".

Illustration 15: The server redirecting the request after trying to modify the cookie without re-signing it

Illustration 16: Cookie Monster finding the secret of the sample application
Using the cookie secret and the file containing the new cookie contents (not base64 encoded), Cookie Monster could be used to create a new session cookie and signature.

Illustration 17: Cookie Monster providing a new session cookie and signing it

Returning back to the Repeater tab of Burp - copying the cookies provided by Cookie Monster into the request and executing it, results in the altered account details being displayed; proving that an authentication bypass is possible.

Illustration 18: A successful authentication bypass using the modified cookie and signature
6. An Example of an Attack on Passport.js Using Local Authentication

A common implementation of Passport.js is to not only offer OpenID strategies, but to also allow users to register and login using a local authentication strategy. A local strategy is one which acts like a traditional authentication system; by handling the data within the local system.

When using local authentication, the developer has complete control over the verbosity level of the data stored within the cookie. Ultimately, they will still be required to store some form of identifier, which with knowledge of the cookie signing secret, can be manipulated just the same as when OpenID is used.

For this example, a small Express application was written which has two users – user1 and user2. Upon logging in as user1, the session and session.sid cookies are created and sent back to the browser; as was seen in section 5.

Decoding the session cookie in the Decoder tab of Burp reveals that this particular implementation of the local strategy stores the user’s ID number in the passport object.
Are Your Cookies Telling Your Fortune?

Using Cookie Monster with the `session` and `session.sid` cookies returned by the Express instance, it is possible to recover the secret.

Once the cookie signing secret is known, it is once again possible to create a new session cookie and sign it; fooling the server into believing it is legitimate.

As the `passport` object in the `session` cookie has one property `[user]` which stores the user ID, the contents of the new cookie need only change the ID from 1 to 2. Once the change to the unencoded cookie is made and stored into a new file (in this case, `new.cookie`), Cookie Monster can be used to encode and sign it.
After setting the `session` and `session.sig` cookies to the new values generated by Cookie Monster, accessing the profile page now provides access to the `user2` account and confirms an authentication bypass.
Are Your Cookies Telling Your Fortune?

Using the form on the profile page, it is possible to update the stored value of the associated name. To verify that any actions carried out with the modified cookie will persist into the back-end, the value “Manipulated Data” was entered into the “New Name” input box and submitted.

After logging out and logging back in as **user2**, the new name is displayed, confirming that full control of the account was acquired.

Illustration 24: Accessing the user2 profile after an authentication bypass

Illustration 25: The user data confirmed as being persistently modified

Hello, Manipulated Data. - **Logout**

See Account Information

www.digitalinterruption.com
7. Summary & Recommendations

Of the 8,186 samples tested, 62 (0.76%) were successfully cracked using the list of secrets gathered using open-source intelligence. This low figure suggests that, at minimum, the hosts tested are for the most part avoiding reusing common secrets.

One of the more alarming findings, throughout the information gathering stage, was how many projects are publicly publishing secrets on the likes of GitHub. As well as a secret utilising a high level of entropy, it’s equally as important that the secret not be leaked.

In order to publish source code publicly and still retain the ability to use signed cookies without compromise, the following steps should be taken:

- Store secrets in either a configuration file or in environment variables
- Don't include default secret values, instead, force the user to set them up during installation
- Re-configure any systems that have had their secret shared on GitHub in the past (committing a new version with it removed does not remove the history stored within Git!)
8. References


7: Express.js, GitHub - expressjs/flash, 2018, https://github.com/expressjs/flash

8: Liam Curry, passport-steam/examples/signon at bf2c3cca044a7aa174182b51fede9f72f147012e · liamcurry/passport-steam, 2018, https://github.com/liamcurry/passport-steam/tree/bf2c3cca044a7aa174182b51fede9f72f147012e/examples/signon
9. Appendix

9.1. Shodan CSV to JSON Script

```javascript
const csv = require('csvtojson')

let servers = []
csv({ noheader: true, delimiter: '||', ignoreEmpty: true })
  .fromFile('servers.csv')
  .on('json', s => {
    let pattern = /session\.sig=(.+?);/
    let sig = pattern.exec(s.field3)[1]
    let data = /session=(.+?);/.exec(s.field3)[1]

    servers.push(
      {ip: s.field1,
       port: s.field2,
       data: data,
       sig: sig}
    )
  })
  .on('done', error => {
    fs.writeFile('servers.json', JSON.stringify(servers), 'utf8', function (error) {
      if (!error) {
        console.log('Exported to servers.json')
      }
    })
  })
```

Are Your Cookies Telling Your Fortune?
9.2. Modifications to app.js

diff --git a/app.orig.js b/app.js
index b0182d9..0ac3643 100644
--- a/app.orig.js
+++ b/app.js
@@ -4,8 +4,8 @@
 var express = require('express')
 , passport = require('passport')
 , util = require('util')
- , session = require('express-session')
- , SteamStrategy = require('../../').Strategy;
+ , session = require('cookie-session')
+ , SteamStrategy = require('passport-steam').Strategy;

 // Passport session setup.
 // To support persistent login sessions, Passport needs to be able to
@@ -52,10 +52,8 @@
     app.set('views', __dirname + '/views');
     app.set('view engine', 'ejs');
     app.use(session({
         secret: 'your secret',
         name: 'name of session id',
         resave: true,
         saveUninitialized: true});
+     secret: 'your secret',
+     saveUninitialized: true});

 // Initialize Passport! Also use passport.session() middleware, to
 support
 // persistent login sessions (recommended).
9.3. Contents of new_cookie.json

```json
{
    "passport": {
        "user": {
            "provider": "steam",
            "json": {
                "steamid": "76561198833340126",
                "communityvisibilitystate": 3,
                "profilestate": 1,
                "personaname": "Rob @ Digital Interruption",
                "profileurl": "https://steamcommunity.com/profiles/76561198833340126/",
                "avatar": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb.jpg",
                "avatarmedium": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb_medium.jpg",
                "avatarfull": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb_full.jpg",
                "personastate": 0,
                "primaryclanid": "103582791429521408",
                "timecreated": 1525104456,
                "personastateflags": 0
            }
        },
        "id": "8675309",
        "displayName": "Jenny",
        "photos": [
            {
                "value": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb.jpg"
            },
            {
                "value": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb_medium.jpg"
            },
            {
                "value": "https://steamcdn-a.akamaihd.net/steamcommunity/public/images/avatars/fe/fe/fef49e7fa7e1997310d705b2a6158ff8dc1cdfeb_full.jpg"
            }
        ],
        "identifier": "https://steamcommunity.com/openid/id/76561198833340126"
    }
}
```